SWITCH-ing from multi-tenant to event-driven videoconferencing services

Jernej Trnkoczy
University of Ljubljana,
Faculty of Civil and
Geodetic Engineering

Uroš Paščinski
University of Ljubljana,
Faculty of Computer and
Information Science
and
Faculty of Civil and
Geodetic Engineering

Sandi Gec
University of Ljubljana,
Faculty of Computer and
Information Science
and
Faculty of Civil and
Geodetic Engineering

Vlado Stankovski
University of Ljubljana,
Faculty of Civil and
Geodetic Engineering
Jamova cesta 2
1000 Ljubljana
E-mail: vlado.stankovski@fgg.uni-lj.si

Abstract—**Full mesh is the most commonly used networking topology in Web Real-Time Communication (WebRTC) based videoconferencing (VC) applications, however, due to its inherently poor scaling capability it is not appropriate for multi-party VC with many participants. Solutions based on centralized media server infrastructures are used to leverage the scaling problem. Service providers adopting centralized approach need to ensure good resource utilization to lower the price, and at the same time provide good Quality of Experience (QoE) to the end users. In practice, even with todays advanced cloud technologies, these two conflicting goals are difficult to achieve simultaneously. In order to tackle this complex problem, we propose an innovative event-driven model, that differs from the traditional multi-tenant service provisioning model. In this work, the architecture and implementation of a WebRTC event-driven multi-party VC, based on Software as a Service (SaaS) principles is presented. A prototype was developed on top of Docker containers and Kubernetes container orchestration technologies, which in our opinion represent key enabling technologies fostering the migration from multi-tenant towards event-driven architectures. The technology readiness to support such time-critical applications is evaluated. The initial results suggest that although there are some trade-offs in terms of performance/resource consumption, our fully functional prototype allows for on-the-fly media server instance creation and destruction in arbitrary cloud provider infrastructure with still acceptable application usability.**

I. INTRODUCTION

With the advent of Web Real-Time Communication (WebRTC)[1], the videoconferencing (VC) applications are becoming widely spread. Any device with a WebRTC-enabled web browser can establish a VC session without additional browser plug-ins. This seamless usage fosters the raise of VC systems, since it is now very easy to include them as a part of Web applications. As a consequence, the number of VC users is growing rapidly.

Many of today's WebRTC applications operate in natively supported mesh network topology, in which every peer in the network sends their data to all other peers in the same VC and requires no media server infrastructure. However, the poor scaling properties of such topology limits the number of participants that take part in a VC call. For multi-party calls with many participants, solutions based on centralised media server infrastructures still represent the only viable option [1]. Traditionally, dedicated hardware servers were used for this. However, with the evolution towards the Web and rapid growth in the number of users, this expensive and non-scalable technology became obsolete.

At the same time, with the development of cloud computing technology, new possibilities in terms of improved scalability have emerged. Computing power on a pay-as-you-go pricing basis has made Videoconferencing as a Service (VaaS) attractive and possible: adaptation to large variations in the number of users became easier. Thus, VaaS providers started to emerge. The challenge for such service provider is to ensure high infrastructure utilisation and at the same time allocate just enough resources to satisfy the Quality of Experience (QoE) requirements of the end users. High infrastructure utilisation and precise resource allocation ensure the lowest possible cost of service (and consequently higher revenue of VaaS providers).

Traditionally, VaaS providers use the multi-tenant application model, in which a single VC application instance is shared by a large number of tenants. For a VC application a tenant is represented by a group of users that have communication needs. In multi-tenant model, simultaneous VC calls of different user groups are served by the same service instance running on dedicated infrastructure. This approach, however, is not ideal because in various cases customer security and privacy concerns may lead to the strict requirement not to share the VC service instance with other tenants. On the other hand, the multi-tenant model allows for high utilisation of physical resources; however, it is still difficult to assure that the VC server will not run in an idle mode. With a rapidly changing number of service clients, ensuring high utilisation becomes more complex and a fine-grained scalability mechanism is needed. Vertical scalability mechanisms may seem a simple solution. However, it turns out that deploying more powerful computing nodes may prove useless as in many cases the scalability bottleneck is in the server network bandwidth and not in the CPU or memory consumption of the server instance [1]. This suggests that the VC service design must

[1] https://www.w3.org/TR/webrtc/

support horizontal scalability as well. Horizontal scaling of multi-tenant media servers is complex, since the application logic of such servers is inherently stateful. Moreover, horizontal scaling within a single data centre may not necessarily improve network-level Quality of Service (QoS) metrics (such as latency, jitter, packet loss and throughput) for the tenants, potentially using the VC service from anywhere.

Therefore, a challenge addressed by this work is to provide a global VC service coverage and the associated problems with assuring appropriate level of networking QoS. Real-time interactive VC applications demand low network delays and sufficient network bandwidth. Purposely built networks work best, but are expensive and limited in range and flexibility. Another option represent guaranteed networking resources, which have to be negotiated with telecommunication operators, but are also expensive, especially for long-distance connections. Our solution is, therefore, to instantiate a VC service for every single usage need, that is closer to the end-users group. Currently, most WebRTC-based applications use the Internet as a best-effort public network that takes care of the traffic management. Appropriate geographical positioning of the VC service in relation to the location of the end-users and the current networking conditions could form basis for achieving higher QoE. This idea is not new by itself. Research in this field goes back to 1997, when dynamic server selection was proposed for the first time [2]. However, possibilities for global service operation have been limited at that time.

In this study we propose a new VC service provisioning model that mitigates various disadvantages of multi-tenant VC applications. Our solution envisions creation of a new service instance for every new VC call by using latest virtualization technologies. This event-driven VC service delivery model could be suitable for companies developing multi-party VC applications and provide for higher QoE to the end-users compared to the traditional multi-tenant approach. The approach, as we further elaborate, has certain technological requirements that cannot be satisfied in traditional Virtual Machine (VM) virtualization environments.

Recent advancements in container-based virtualization could be used to address the technological requirements (such as fast VC service provisioning) and could actually make an event-driven architecture a reality. The purpose of our ongoing work is therefore to investigate the key prospects and limitations of an even-driven architecture by designing, developing and evaluating a real-world WebRTC multi-party VC system following the event-driven approach. In this work, an evaluation of appropriateness of the Docker container ecosystem for realizing the event-driven approach is presented. An initial evaluation of a developed prototype application is also provided, while detailed evaluation is still ongoing. It is important to note that the precise cloud resource allocation mechanisms for a particular service instance are not in scope of this study, and are left for future work.

II. Related work

With rising popularity of virtualization technology and cloud computing, a lot of research has focused on evaluation and comparison of different SaaS architectures. Multi-tenant and multi-instance are two widely recognized approaches for the delivery of SaaS. The authors [3] discuss the benefits and shortcomings of the multi-tenant SaaS delivery model from the service provider viewpoint. They find that benefits outweigh the challenges related to complexity of multi-tenant applications. Similarly, the work [4] concludes that multi-tenant is the preferred model; however, they also mention the challenges that are mostly related to resource isolation, scalability, security, downtime and maintenance. Another work [5] also finds that security and related resource isolation is a critical point in multi-tenant applications. Compared to these, the paper [6] discusses that the multi-instance approach, while being easier to implement, is better suited, if the number of tenants is likely to remain low. They state that the multi-instance model suffers from an increased maintenance cost, which can in part be attributed to the effort for deploying updates to numerous instances of the application. The above mentioned papers are mostly theoretical and do not take into account the specifics of particular application. The paper [7] provides empirical data showing lower operational costs for multi-tenant SaaS. However, they warn that this may not apply to all SaaS applications, since the suitability of the multi-tenant model is application dependent. They conclude that the software industry is constantly improving and that SaaS providers might find a better way to deliver their services exploiting a multi-instance application arrangement.

All of the above research efforts focus on the achievement of cost-effective solutions for cloud service providers (for SaaS in general, not for specific applications), and all of them propose multi-tenant architecture as a general solution. It is therefore not unusual that current VaaS providers follow this approach. However, with the gradual transition to mobile and Web applications and the corresponding growth in demand of VC applications the providers are facing two key challenges:

1) how to achieve efficient, cost-effective scaling of computing and networking resources, and
2) how to achieve high QoE.

Since vertical scaling of multi-tenant server instance cannot solve these problems efficiently, most of the work has focused on the horizontal scalability and systems with many geographically distributed servers, serving the same application instance. The efforts in this field differ depending on the type of media servers used (e.g. mixers as Multipoint Control Units (MCUs) or routers as Selective Forwarding Units (SFUs)), the particular sub-problem they want to address (e.g. ensuring networking QoS, reducing the Wide Area Network complexity, scaling of computing resources and so on, and combinations of these), and the communication topology they adopt.

With regard to topology, horizontal scaling and geographic distribution of servers can be realized by using:

1) a centralised topology, in which users belonging to the same VC call always communicate through a single server instance, and
2) a multi-server topology, in which users belonging to the same VC can be attached to different geographically dispersed server instances.

In the case of a centralised topology, scaling is achieved at the level of the VC call. Although the VaaS system could be composed of many servers, the communication flow of participants belonging to a particular VC session never goes through different servers. A VC session has to be managed by a single media server and it can never be split into two or more server instances. Therefore, the highest scaling granularity that one can achieve is that of a VC session.

In the case of a multi-server topology the VaaS system contains multiple distributed media servers and the users belonging to the same VC call are usually connected to different media servers. Participants send their media streams to the optimal media server in a cloud environment. The media servers aggregate the streams and send them to other media servers that are optimal for other users participating in a VC session.

In order to satisfy QoS and QoE optimisation goals, regardless of the topology adopted, and by knowing the client locations and the number of servers constraint, the key question is the same: where the servers should be placed and how clients' streams should be mapped to servers. Optimisation goals could, for example, aim to reduce the overall end-to-end latencies between the clients, maximise the bandwidth usage and the processing resources for tasks off-loaded from clients to servers and similar. However, in contrast with the centralised topology, the optimisation problem is much more complex in the case of a multi-server topology.

We believe that providing a cloud provider independent solution is more favourable than currently existing vendor-locked or proprietary solutions. Our solution allows for SaaS providers to operate a Software Defined Data Centre making use of many geographically distributed cloud providers. In such a Software Defined Data Centre, it is possible to imaging VC services starting and stopping for every single usage need. To the best of our knowledge, no solution so far has proposed such event-based VC system or implemented a prototype with similar capabilities. This work presents its design and implementation, while a detailed evaluation study is still ongoing.

III. Technological requirements

The VC application is a very particular type of application, in which several participants (clients) are engaged in a short-lived conversation. Therefore, the event-driven approach is very suitable for VC application (e.g. pay only for the time-slot and amount of resources you need). On the other hand, a VC is an interactive real time service with stringent QoS requirements that depend on guaranteed computational and network resources. While various private and public clouds can offer a certain degree of networking QoS inside their cloud environments (technologies like Open vSwitch, OpenFlow, MPLS, etc.), guaranteed networking QoS in the open Internet is still impossible. Hence, to address QoS requirements of the VC application, the proper selection of the geographical location of the VC host machine, targeting the specific group of end-users (clients) is considered the best strategy. Therefore, the proposed event-driven approach poses certain technical requirements, such as:

- Service instance creation time that should be short enough to be tolerable by the users.
- Availability of a fine-grained on-demand infrastructure leasing mechanism. For example, the pricing of resources consumed by the service instance should be on a pay-as-you-go basis and it should allow for pay-per-minute model.
- Guaranteed compute resource reservation. The real-time VC applications need guaranteed memory and CPU resources that are well isolated from the other users sharing the infrastructure.
- Resource reservations with fine granularity. The service instance should be allocated exactly the amount of resources required to support the established VC session.
- Possibility to start the service instance globally and in arbitrary location, which is motivated by: (i) the need for global instance management/control that supports automatic service creation anywhere in the World, and (ii) the need for mechanism/technology that supports multiple clouds and seamless portability between any cloud provider and infrastructure type. This assures the availability of infrastructure practically in arbitrary location, worldwide.
- Highly-automated and efficient services administration/orchestration. Managing large number of services can become costly, if it is not done efficiently.

The possibility to fulfil these requirements became possible with the recent developments in container based virtualization technologies (e.g. Docker and LXC) and container orchestration tools (e.g. Kubernetes, Mesos and Swarm). The later might prove to be key enablers for the migration from multi-tenant to event-driven applications.

IV. Implementation

Achievement of fully functional event-driven VC application, which adheres to the above requirements, required integration of several technologies. To better understand the design, workflow, technology specifics and their benefits as well as limitations, all the key technologies that were used for the present study are discussed in the following subsection.

A. Baseline technologies

1) Jitsi-meet as WebRTC VC software: Centralized units used in multy-party VC systems that do not follow a pure peer-to-peer approach are of two types: (i) MCU and (ii) SFU[2]. Both offer mechanism for enabling and managing group

[2]https://tools.ietf.org/html/rfc7667

communications through a centralized component and both can be used in WebRTC-based systems. However, there is a fundamental difference in their design.

The MCUs transcode, mix and multiplex different audiovisual streams into a single one. By doing so, their advantage is optimized bandwidth consumption. However, the composite mixing requires a decoding of all the input streams, a mixing process, and an encoding process, which altogether tend to increase the latency. Additionally, the composition is CPU intensive processes, which makes the scaling to a large number of participants impossible.

Selective Forwarding Unit (SFU) are centralized units that perform only forwarding/routing of video streams. They forward the incoming media stream from participant to outgoing media streams to be received by other participants. This component only forwards real time protocol packets, optionally changing their headers, but without processing the payload. The forwarding mechanism, which forwards incoming stream to receivers, can be controlled by the application. Unlike MCUs, the advantage of SFUs is lack of need for heavy processing, because they do not perform transcoding and mixing. Additionally, without encoding/decoding, the latency of the added SFU media server is minimal. Lastly, the clients with full correspondence with the SFU media server have complete control over the streams they receive, and because the clients receive the streams they want, they can have full control over the user interface flexibility. However, SFUs have the "least common codec" disadvantage, in which every participant in the conference needs to use the same codec. Another disadvantage is the need for higher bandwidth and processing power of the participants (e.g. if there are N participants, any of them wishing to visualize the video of the whole group needs to receive and decode N-1 video streams). The latter can be mitigated by using a selection algorithm to decide which packets should be forwarded and to which endpoints (e.g. last N configuration).

To support the above discussion, in Section V it will be shown that while SFU-based WebRTC services still require substantial amount of CPU power, compared to MCUs they represent an attractive approach to address the server performance issue and at the same time offer maximum flexibility for the client User Interface (UI). With the raise of bandwidth and processing capabilities of clients (which could be a limiting factor when using SFU) the SFU became very popular in WebRTC cloud-based systems. This is why we chose to base our system on the SFU. We selected the Open Source project Jitsi-meet to develop our prototype.

The Jitsi-meet is a WebRTC based multi-party VC software with a production-level quality, while all the constituting components are Open Source.

The complete Jitsi-meet prototype system is composed of four server-side components:

1) Jitsi Videobridge is an SFU unit responsible for controlling/forwarding video/voice media streams between participants.
2) Jicofo acts as a conference focus, taking care of managing the videoconferences, the signalling needed to establish WebRTC connectivity, terminating the calls, etc.
3) Prosody is the XMPP server allowing the exchange of the signalling messages.
4) Web Server serves the JitsiMeet – a JavaScriptWebRTC application to the participants.

In the course of our work, for experimentation we implemented the components in two ways: (1) all components as a single Docker container and (2) each of the four components as a separate Docker container.

2) Docker as container-based virtualization: Containerization is the process of distributing and deploying applications in a portable and predictable way. It accomplishes this by packaging components and their dependencies into standardized, isolated, lightweight process environments called containers. These are not new concepts, with some operating systems leveraging containerization technologies for over a decade. For example, LXC, the building block that formed the foundation for later containerization technologies was added to the Linux kernel in 2008. It combined the use of kernel control groups (allows for isolating and tracking resource utilization) and namespaces (allows groups to be separated) to implement lightweight process isolation.

Containers come with many attractive benefits for developers and system administrators, such as: (i) abstraction of the host system away from the containerized application, (ii) seamless scalability, (iii) simple dependency management and application versioning, (iv) lightweight execution environments and others. Comparing to VM virtualization, containers provide a lighter execution environment, since they are isolated at the process level, sharing the host's kernel. This means that the container itself does not include a complete operating system, leading to very quick start-up times and smaller transfer times of container images.

Among several containerization technologies available today, Docker is the most common containerization software in use. While not introducing many new ideas, Docker made containerization technologies accessible by simplifying the process and standardizing on an interface. It was developed to simplify and standardize deployment in various environments. By packaging up the application with its configuration and dependencies and shipping as a container image, the application will always work as designed locally, on another machine, in test or production. Moreover, Docker containers spin up and down in seconds, improving horizontal scalability at any time to satisfy peak customer demand, when needed.

Docker containers are launched from Docker images. An image can be basic, with nothing, but the operating system fundamentals, or it can consist of a sophisticated pre-built application stack ready for launch. Docker images follow the principle of shared layering: each image consists of a series of layers. When building images with Docker, each command executed forms a new layer on top of the previous one. These layers can then be reused to create new images. Docker makes

use of union file systems to combine these layers into a single image. Union file systems allow files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system. If multiple containers are based on the same layer, they can share the underlying layer without duplication, leading to optimized disk space utilization. Perhaps even more important aspect of layering is image transfer time. By making use of layers cache, it is possible to reduce transfer times and required bandwidth, because what is actually transferred are image layers that are later on combined to make up the image itself. If most of the layers constituting a certain image are already present on the machine where the image is needed, the transfer will be fast, since only the missing layers will be transferred.

3) Container orchestration with Kubernetes: Docker provides all of the functions necessary to build, upload, download, start, and stop containers. It is well-suited for managing these processes in single-host environments with a relatively small number of containers. When it comes to orchestrating a large number of containers across many different hosts, Docker tools are not sufficient. Clustered Docker hosts present special management challenges that require a different set of tools, usually called orchestration tools or schedulers. When applications are scaled out across multiple host systems, the ability to manage each host system and abstract away the complexity of the underlying platform becomes necessary. Container orchestration is a broad term that generally refers to cluster management and container scheduling. The orchestrator represent the primary container management interface for administrators of distributed deployments. Currently, several orchestration tools capable of managing distributed deployments based on Docker containers exist. For example, Kubernetes[3], Docker Swarm[4], Mesos/Marathon[5], etc. These tools offer management, scheduling and clustering capabilities that provide the basic mechanisms for scalability and controllability of container-based applications, and vary in capabilities.

4) Fabric8 Kubernetes client: Kubernetes cluster should be a relatively self-contained unit. This is not strict requirement; however, due to the design of Kubernetes scheduling and network routing domains, each cluster should be arranged within a relatively performant, reliable and cheap network, which in practice confines the cluster boundaries to a single data centre or single availability zone of a cloud provider. The global VaaS, on the other hand, should be running over different cloud providers and multiple data centres and availability zones. Therefore, in our scenario the VC services should be scheduled across multiple clusters. In order to allow VC instance creation over several clusters, a software capable of connecting to different clusters and scheduling services on them is needed.

Kubernetes cluster exposes Kubernetes API on master nodes. This API provides internal and external interface to Kubernetes cluster. Fabric8 Kubernetes client is a Java library providing simplified access to the Kubernetes API. Although Kubernetes RESTful API can be accessed directly, the use of Fabric8 Kubernetes API allowed us to abstract away the complexities related to security aspects (authentication, authorization) and asynchronous nature of Kubernetes API, which is challenging to support using request-response HTTP/HTTPS protocols. Fabric8 Kubernetes API Java libraries were therefore used in our VC instance management layer, which was implemented as Java Servlets running in Apache Tomcat Server container. This VC instance management layer is responsible for adding and removing application component instances, managing the amount of RAM and CPU share of individual containers and determining the subset of hosts or individual host, where the container will be placed.

B. Application prototype and testbed

The overall architecture of the prototype is depicted in Figure 1. Seven geographically distributed clusters formed an initial, geographically widely distributed testbed. As previously mentioned, the Jitsi-meet software was implemented in Docker container images that can be managed by the Kubernetes container orchestrator tool. A Web application for the management of application instances was developed as a Java Web based API that uses the Fabric8 libraries to access the Kubernetes clusters. A decision, which components would be shared between multiple tenants and across multiple VC sessions and which components would be instantiated per single VC session was made. For example, the signalling components that are needed to establish VC sessions, and are not computationally and bandwidth demanding are shared by all the customers of the VaaS service. However, we decided to isolate the VC sessions and instantiate for each VC call both signalling and media services. In summary, in the developed prototype, for each new VC session all four Jitsi-meet components are instantiated as Docker containers. Due to the architectural design of the Jisti-meet application all four component instances serve only one VC session and they are deployed together on the same host machine – the four Docker containers are scheduled on Kubernetes as a single Kubernetes pod. To expose the instantiated services outside the cluster, an appropriate Kubernetes proxy services also need to be deployed.

Following is a sequence of planned interactions between a user requesting high QoE VC session and the event-driven VC service.

1) A conference organizer wants to create a VC session. By using the Web application in the browser, all the actions are reflected as HTTP REST-based request. To maximize the user experience (QoE/QoS), the Web application performs a context capturing phase. For example, the organizer could type the IP addresses of the conference participants or obtain their geolocations through other services (e.g. Google services), determine the VC application instance configuration (last-N, simulcasting etc.) and other more automatized mechanisms.

[3]https://kubernetes.io/

[4]https://docs.docker.com/engine/swarm/

[5]http://mesos.apache.org/

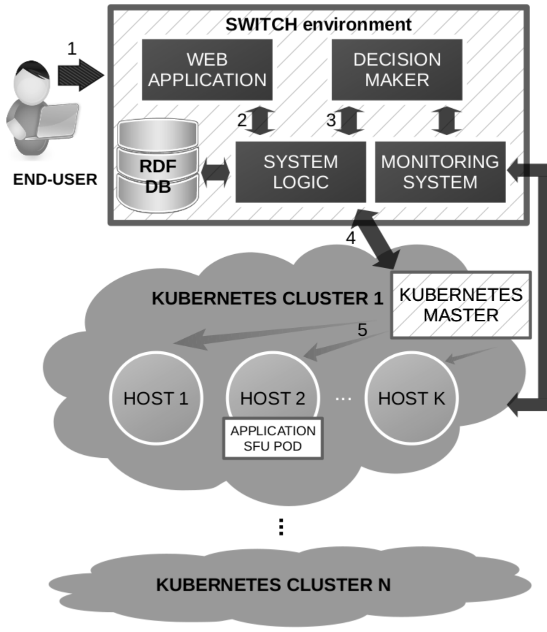

Fig. 1. Architecture prototype showing the main steps of deploying/undeploying the container-based Jitsi-meet VC application.

2) The Web application extracts the context, which is then forwarded to a Decision Maker service, that decides the best geographical position for the VC service session (based on an implemented QoE model, e.g. minimal average latency between the clients and the potential location of the new service) to be deployed and also determines the precise amount of hardware resources (memory and CPU power) that are needed for that particular VC session. All the metadata of the system is stored in a Knowledge Base, while the monitoring data needs to be stored in a purposive storage such as Time Series Database (TSDB).
3) Decision Maker service sends notification on a selected Kubernetes cluster host machine to the System Logic service.
4) The System Logic service initiates the creation of a Kubernetes service pod, in our case the Jitsi-meet application, on the chosen Kubernetes cluster host machine. The request for the pod creation is sent to the appropriate Kubernetes Master, which asynchronously creates the pod.
5) During the pod creation process, the end-user is notified about the progress. After application is deployed and running, its respective URL is provided to the end-user.

Upon successful deployment of the application, the VC organizer and the participants can enjoy the VC session. When the session is finished, the organizer simply triggers the stop event that undeploys the pod and frees all the allocated resources of the session. This action is also asynchronous, so the progress is shown to the organizer with the final notification among all the participating users.

V. Evaluation

A. Discussion on technology readiness

1) Benefits of container-based virtualization (why containers are better than VM): The benefits of container-based virtualization in relation to traditional VM virtualization has already been discussed by Pahl et al. [8], who found benefits of container-based virtualization, such as quick startup time and convenient packaging. In the following we provide a qualitative analysis of the developed prototype and evidence for its functioning according to the requirements presented in Section III. More detailed evaluation and experimentation is still ongoing.

Quick startup time: Our proposal of event-based approach assumes creation of a new media server instance for every new VC call. However, booting up a VM often takes time and does not complete fast enough; thus, keeping VMs in disk and booting VMs on demand is often not practical. This is why in traditional VM-based clouds providers use multi-tenancy, which allows the unit of sharing to be smaller than a VM and thus enable more sharing of resources.

Good resource isolation allowing for guaranteed resources: Traditional multi-tenant approach, in which, for example, one application instance serves thousands of users, has difficulty of efficiently separating resources to maintain QoE among all the users. In our case, event-driven one-instance-per-VC-call approach means that we have pushed multi-tenancy from application layer to the infrastructure layer, since a new separate container instance is created and destroyed on demand for every new VC request [9], that generally leads to better resource isolation.

Pay-as-you-go pricing with high temporal granularity: Our proposal of event-based multi-instance approach assumes creation of a new media server instance for every new VC call. Because VC sessions are typically short-lived, reaching acceptably low service cost requires IaaS infrastructure that allows to lease instance only for a needed time period. For example, Amazon's Elastic Compute Cloud (EC2), offers on-demand VM instance, which lets users pay for compute capacity by the hour with no long-term commitments. Container management (start/stop) allows IaaS providers to decrease the temporal granularity of leasing, for example Joyent Triton Compute cloud infrastructure[6] already offers per minute billing for containers.

Low cost of instance management: With the proposed event-driven approach we can achieve thousands of individual services running independently from each other. Highly-automated services administration/orchestration (port management, IPs management, etc.) is required. While the high operational costs of running thousands of instances (multi-instance approach) might seem prohibitive, in the future we believe that with rapid evolution and widespread adoption of

[6]https://www.joyent.com/triton/compute

container-based technologies the SaaS instance management and application upgrade costs will decrease. Thus, it makes the instance-based approach the preferred choice for SaaS providers.

No vendor lock-in: Another notable problem of VM-based virtualization is vendor lock-in, because it is not straightforward to migrate a VM image created within one cloud provider to other cloud provider. However, container based technologies and container orchestration technologies avoid vendor lock-in for multi-cloud deployment; although, different container technologies might not be jointly compatible, which introduces platform dependency.

High performance: The ability of Docker to work closer to bare-metal brings performance advantages, relevant for real time services that, like WebRTC, are very sensitive to. From the perspective of VaaS provider, which is leasing infrastructure from public cloud, this might not be a direct requirement. For example, in [10] authors made an up-to-date comparison of the Docker containers and VMs hosting a WebRTC application and they confirmed that the overhead of VMs affects the performance by 5–10%.

2) Issues with container-based virtualization:

Problems with compute resource allocation in Kubernetes: While Kubernetes provides resource allocation, its built-in scheduler might neither take good decisions in case of heterogeneous machines in a cluster nor adequately address current load on the cluster nodes.

Performance regression of overlay networks: Network virtualization is a very complex and problematic topic when trying to achieve optimal QoE in WebRTC cloud deployments. With virtual networks, some or all of the hardware components such as network cards, switches, routers, etc. are replaced with respective software counterparts. Besides, TCP/IP stack might need to be extended with additional layer embedded in the application layer, inducing lower ratio between the size of payload versus the size of headers. Some might require presence of distributed key-value store for management. Virtual networks are still less performant compared to hardware counterparts, resulting in higher delays, lower bandwidth and higher packet loss, while also utilising more computational resources.

Lack of bare-metal container providers: Currently, most cloud providers offer containers on top of VMs, which leads to double virtualization. While rental of physical machines and installation of Docker on top of them is possible, it is more expensive and less flexible option compared to virtual machines. One of the few providers offering containers on bare-metal is Joyent with their Triton Elastic Container Infrastructure, albeit their container technology, while being compatible with Docker interfaces, relies on different implementation. Clearly, double virtualization can be considered redundant, at least from a performance perspective.

Weaker isolation of containers compared to VMs: Escaping from container to underlying host or to another container is easier than with VMs, which implies weak support for multi-tenancy, but in our case container clusters as well as containers are maintained by a single SaaS provider.

B. Experimental results

The evaluation environment consists of seven cloud clusters spread among different locations around the World: flexiOps nodes are located in Edinburgh (UK), Arnes nodes in Ljubljana (Slovenia), Google Cloud Platform (GCP) US West in The Dalles Oregon (USA), GCP EU West in St. Ghislain (Belgium), GCP Asia East cluster in Changhua County (Taiwan), GCP Asia South-East in Jurong West (Singapore), and GCP Asia North-East in Tokyo (Japan). Their characteristics roughly match with only few minor differences: all VMs utilise a single vCPU, except flexiOps machines which are run by four. Arnes and GCP clusters run on Intel Xeon processors, ranging from Ivy Bridge, Broadwell and Haswell platforms, while the flexiOps cluster is powered by ageing AMD Opteron processors. vCPUs are scheduled on a pre-emptive basis for Arnes and flexiOps, but are dedicated per hardware thread for GCP VMs. Nodes have 3.75 or 4 GB of RAM and 80 or 100 GB of disk space backed by hard disks shared over network. Arnes' and flexiOps' Docker networking is based on overlay network (flannel), while GCP avoids overlay with network over dedicated hardware switch. Docker versions are 1.10.3 for Arnes and 1.11.2 for the rest of the nodes.

1) Compute resources consumption: In this experiment a stress test was made for all available Kubernetes clusters by using Jitsi-hammer[7] software. The Jitsi-meet application was run in one of the Kubernetes nodes of the cluster whereas the Jitsi-hammer was run on the other node in the cluster. To improve performance, Kubernetes logging was disabled, as by default GCP collects all the output printed to standard or error output to Elastic search database, which can hinder the performance of the containers if a lot of log content is generated. Despite this optimisation, Jitsi-meet video bridge component was able to saturate the CPU completely when merely serving 18 users. We claim that the VM was a bit too weak for the task. The results for GCP US West are depicted in Figure 2.

2) Frame delay among the clouds: Another experiment focused on more specific VC related metric – video roundtrip. The measurements are depicted in Figure 3.

VI. Conclusion

Multi-tenancy has been so far the predominant delivery model for cloud services. However, with the emergency of container based technologies, it is now possible to orchestrate services seamlessly across software defined data centres. This allows us to switch from multi-tenant to event-driven services architecture, supporting diverse applications including complex applications such as VC. In the course of this work we developed an event-driven VC service prototype based on Docker container virtualization and Kubernetes container orchestration technologies.

[7]https://github.com/jitsi/jitsi-hammer

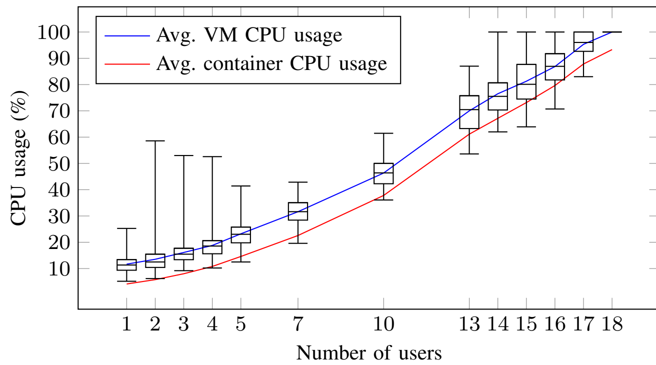

Fig. 2. Average CPU usage of VM (blue) and its container (red) during Jitsi-meet running in Kubernetes cluster deployed in GCP US West region over 120 runs. Box plots show the median VM's CPU usage, and the first and the third quartile. Whisker plots show the minimum and the maximum VM's CPU usage. VM's CPU was maxed out for 18 users participating simultaneously in a VC.

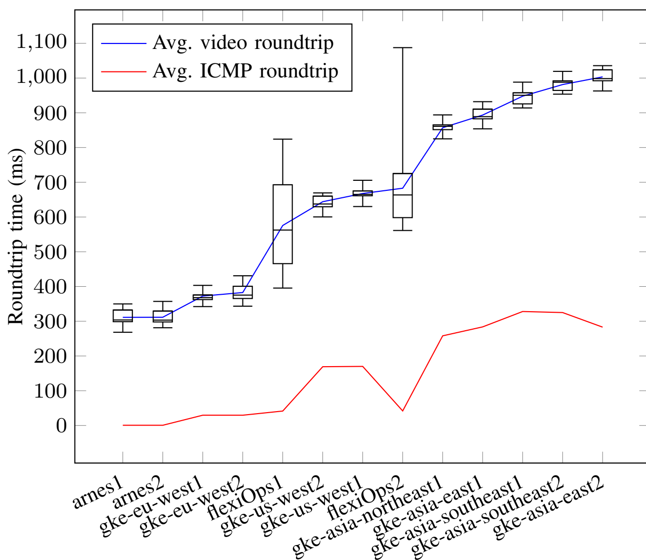

Fig. 3. Roundtrip time measurements performed from a Jitsi-meet client located in Ljubljana against all of the available Kubernetes clusters and machines within. The measurements were done by using ping and VideoLat software. As expected, the geographical proximity plays important role in the VC QoS.

The realisation of event-driven VC services is particularly challenging since the QoE of real time VC applications depends on many environmental conditions, including network properties, which all have to be considered when making decisions on resource allocation and VC services placement for the particular users. This paper presented functional and non-functional requirements of the event-driven VC applications, and technological considerations for the realisation of the event-driven architecture. Furthermore, the paper presented the measurements of parameters that are relevant for the evaluation of the event-driven VC applications. The CPU consumption with varying number of users was presented. It is important to note that the estimation of the needed computational resources is challenging, since it also depends on the configuration (e.g. last-n and simulcasting parameters) of the VC application. The presented video roundtrip measurements prove our assumption that the selection of the instance geolocation is very important to achieve lower video delays, which is besides network throughput important factor of the end-user QoE. Further experimental research is needed to propose the exact algorithms for geolocation selection and precise computational resource allocation. In our ongoing research work we are trying to exploit the Knowledge Base and inference mechanisms, coupled with monitoring system to achieve this. However, in this paper we proved on a real application prototype that, in case of appropriate selection of service instance location and precise computational resources allocation, our event-driven architecture provides cost-effective and highly scalable solution, that at the same time mitigates the vendor lock-in problem and increases the privacy of the service users.

Acknowledgement

This project has received funding from the European Unions Horizon 2020 Research and Innovation Programme under grant agreement No. 643963 (SWITCH project: Software Workbench for Interactive, Time Critical and Highly self-adaptive cloud applications).

References

[1] Y. Lu, Y. Zhao, F. Kuipers, and P. Van Mieghem, "Measurement Study of Multi-party Video Conferencing," in *Proceedings of the 9th IFIP TC 6 International Conference on Networking*. Chennai, India: Springer-Verlag Berlin Heidelberg, 2010, pp. 96–108.

[2] R. L. Carter and M. E. Crovella, "Server selection using dynamic path characterization in wide-area networks," in *Proceedings of the IEEE INFOCOM '97*, Kobe, Japan, April 1997.

[3] R. Krebs, C. Momm, and S. Kounev, "Architectural Concerns in Multi-Tenant SaaS Applications," in *Proceedings of the 2nd International Conference on Cloud Computing and Services Science (CLOSER 2012)*, Setubal, Portugal, 2012.

[4] C. Bezemer and A. Zaidman, "Multi-tenant SaaS applications: maintenance dream or nightmare?" in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*. ACM, September 2010, pp. 88–92.

[5] M. Pathirage, S. Perera, I. Kumara, D. Weerasiri, and S. Sanjiva Weerawarana, "A scalable multi-tenant architecture for business process executions," *Web Services Research*, vol. 9, no. 2, pp. 12–41, 2012.

[6] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao, "A framework for native multi-tenancy application development and management," in *Proc. Int. Conf. on E-Commerce Technology (CEC) and Int. Conf. on Enterprise Computing, E-Commerce, and E-Services (EEE)*. IEEE, 2007, pp. 551–558.

[7] H. A. and S. F., "Cloud Computing: A Multi-tenant Case Study," in *Kurosu M. (eds) Human-Computer Interaction: Users and Contexts. HCI 2015. Lecture Notes in Computer Science*, vol. 9171. Springer, Cham, 2015.

[8] C. Pahl and B. Lee, "Containers and Clusters for Edge Cloud Architectures a Technology Review," in *3rd International Conference on Future Internet of Things and Cloud*. IEEE Computer Society, 2015, pp. 379–386.

[9] C. Momm and R. Krebs, "A qualitative discussion of different approaches for implementing multi-tenant saas offerings," in *Software Engineering*, 2011.

[10] C. C. Spoiala, A. Calinciuc, C. O. Turcu, and C. Filote, "Performance comparison of a WebRTC server on Docker versus Virtual Machine," in *13th International Conference on Development and Application Systems*. Suceva, Romania: IEEE, 2016, pp. 295–298.